#17
T.D.
06/09/04

IN THE UNITED STATES
PATENT AND TRADEMARK OFFICE

Applicant:	**Guo et al.**	Case:	**Guo 3-3-2-22-2**
Serial No.:	**09/538,351**	Filed:	**March 29, 2000**
Examiner:	**David E. England**	Group Art Unit:	**2143**

Title: **METHOD AND SYSTEM FOR CACHING STREAMING MULTIMEDIA ON THE INTERNET**

Mail Stop RCE
COMMISSIONER FOR PATENTS
P.O. Box 1450
Alexandria, VA 22313-1450

SIR:

DECLARATION UNDER 37 C.F.R. § 1.131

We, Katherine H. Guo, Markus A. Hofmann, and Sanjoy Paul, hereby declare as follows:

1. We are Applicants of the above-captioned patent application.

2. We conceived of the complete invention as claimed in the above-identified patent application on or before January 24, 2000. In addition, due diligence toward reducing the invention to practice was exercised from the conception date of the complete invention, as well as the various portions thereof, to a subsequent constructive reduction to practice of the invention.

3. To establish the conception date of the invention disclosed in the above-identified application that predates the earliest effective filing date of U.S. patent 6,484,199, issued November 19, 2002 to Eyal, (hereinafter referred to as the "Eyal '199 patent," having an effective filing date of January 24, 2000), Applicants submit an invention disclosure form for the above-identified invention that was prepared by the inventors prior to January 24, 2000. Applicants' conception date precedes the effective filing date of the Eyal '199 Patent. A subsequent constructive reduction to practice of the invention occurred on March 29, 2000, with the filing of the above-identified application.

The undersigned, Katherine H. Guo, Markus A. Hofmann, and Sanjoy Paul, hereby declare that all statements made herein of our own knowledge are true and that these statements made on information and belief are believed to be true and further that these statements were made with knowledge that willful false statements and the like so made are punishable by fine or imprisonment, or both, under Section 1001 of Title 18 of the United States Code and that such willful false statements may jeopardize the validity of this application or any patent resulting therefrom.

5/14/2004 Katherine Guo
Date Katherine H. Guo

5/17/2004
Date Markus A. Hofmann

5/14/2004 Sanjoy Paul
Date Sanjoy Paul

Eamon J. Wall
Moser, Patterson & Sheridan
Attorneys at Law
595 Shrewsbury Avenue, Suite 100
Shrewsbury, New Jersey 07702

Lucent Technologies
Bell Labs Innovations

Bell Laboratories

Subject: Differential Security for VPNs

Date:

From: A. V. Aho
Org. BL01134
HO 4E-608
(732) 949-9442

B. Freedman:

Attached for patent consideration is a patent disclosure by Katherine Guo, Markus Hofmann, Sanjoy Paul, T. S. Eugene Ng, and Hui Zhang. It describes a novel architecture for caching streaming multimedia on the Internet by extending the existing caching systems. It proposes a set of cache placement and replacement policies specially designed for streaming media.

Traditional solutions for streaming multimedia on the Internet do not scale in terms of object size and number of supported streams. While caching is the standard technique for improving scalability, existing caching schemes do not support streaming media well. We study techniques to enhance caches to better support streaming media over the Internet. We consider an architecture in which *helper* machines inside the network implement several new functions specifically designed to support streaming media. These include segmentation of a media object into smaller units, cooperation of helper machines, and novel placement and replacement policies for segments of media objects. We discuss both the architectural and system issues of implementing these functionalities.

Alfred V Aho

A. V. Aho

HO-BL01134-AVA-KKS-dks

Cache Placement and Replacement Policies for Streaming Multimedia on the Internet

Katherine Guo, Markus Hofmann, Sanjoy Paul
Bell Laboratories
101 Crawfords Corner Road
Holmdel, NJ 07733, USA
(kguo,hofmann,sanjoy)@bell-labs.com

T. S. Eugene Ng, Hui Zhang
Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA 15213, USA
(eugeneng,hzhang)@cs.cmu.edu

Abstract

Existing solutions for streaming multimedia on the Internet do not scale in terms of object size and number of supported streams. While caching is the standard technique for improving scalability, existing caching schemes do not support streaming media well. We study techniques to enhance caches to better support streaming media over the Internet. We consider an architecture in which *helper* machines inside the network implement several new functions specifically designed to support streaming media. These include segmentation of a media object into smaller units, cooperation of helper machines, and novel placement and replacement policies for segments of media objects. We discuss both the architectural and system issues of implementing these functionalities.

1 Introduction

Internet and World-Wide-Web are becoming the ubiquitous infrastructure for distributing all kinds of data and service, including continuous streaming data such as video and audio. Last several years have witnessed a significant development of commercial products for playback of stored video and audio over the Internet [7, 6], and proliferation of server sites that support audio/video contents. However, existing solutions for streaming multimedia on the Internet have several shortcomings because they use a separate unicast stream for each request, thus require a stream to travel from the server to the client across the Internet for every request. From the content provider's point of view, the server load increases linearly with the number of receivers. From the receiver's point of view, she must endure high start-up latency and unpredictable playback quality due to network congestion. From the ISP's point of view, streaming multimedia under such an architecture poses serious network congestion problems.

Caching is a common techniques for enhancing the scalability of general information dissemination systems. However, it can be directly applied to support streaming media playback over the Web. Caching has been extensively used in the Web context for reducing network load, server load, and access latency. Surprisingly, there has been very little work to extend cache systems to support streaming media. Existing caching schemes are not designed for and do not take advantage of streaming characteristics. Video objects are usually too large to be cached in their entirety. A single, two hour long MPEG movie, for example, requires about 1.4 Gbytes of disk space. Given a fixed investment on buffer space, only a few streams could be stored at a cache, thus, decreasing the hit probability and the efficiency of the caching system. A natural solution would be to break video objects into smaller chunks for the purpose of caching. However, existing caching systems will treat different chunks from the same video object independently, while it might be desirable to consider the logical relationships among the chunks.

In this work, we explore the following techniques to enhance caching systems to better support streaming media over the Internet, namely *Segmentation of Streaming Objects*, *Cache Placement and Replacement Policies*, and *Cooperation of Caches*. We then define a new architecture that realizes these techniques taking advantage of unique properties of streaming media.

The key components of the architecture are so-called *helper* machines, which are caching and streaming agents inside the network, communicating via a novel scalable state distribution protocol and directing requests to the most appropriate helpers.

2 Existing Caching Systems for Web Objects

2.1 Cache Placement and Replacement Policies

In general, cache placement techniques fall into the following three categories:

1. Server push: Server periodically multicasts popular objects to some or all caches in the caching system to report content changes on the server.

2. Cache pull: Each cache individually decides on which objects are popular and requests them from the server.

3. Demand driven: Clients request web objects from certain caches. The caches that serve the requests cache the objects along with passing them to the clients.

These techniques can be used independent of each other and common web caching systems use a combination of them.

If the disk space is full when new objects need to be cached, existing contents in the cache should be replaced to make room for new contents. Common cache replacement policies include Least Recently Used (LRU) and Least Frequently Used (LFU).

2.2 Caching System Organization and Locating the Right Cache

Caching of web-objects for improving end-to-end latency and for reducing network load has been studied extensively starting with CERN httpd [1]. Novel research was done in *hierarchical caching* as a part of the Harvest project [2, 4]. In Harvest, the caches are *statically* configured in a hierarchy such that on a cache miss the request is forwarded to the parent cache. The obvious drawback of such a system is the additional latency incurred in going up the hierarchy when the requested object could be present in a neighboring (sibling) cache. This idea has been exploited in Squid [8] caching architecture and is called *co-operative caching.* However, even in Squid, the neighbors (or siblings) of a cache are *statically* configured using a configuration file. That is, when a cache miss occurs, a cache checks with each one of its statically configured neighbors to see if the requested object is stored there. Another way of stating this is that a cache always checks the same set of neighboring caches for any missing object.

3 Streaming Extensions for Caching Systems

Current caching systems are restricted to support *static objects*, such as HTML pages or images. Static objects are relatively small and they are always cached in their entirety. This design does not properly support *streaming objects* like video and audio clips. Streaming objects consist of data whose transmission has temporal characteristics such that the transmission rate is explicitly regulated or else the data becomes useless (In this study, we assume the transmission rate of streaming objects to be a constant bite rate (CBR). However, the described techniques can be extended to also support streaming objects with variable bit rate (VBR)). In addition, the size of streaming objects normally is at least an order or two larger than that of static objects.

Regardless of whether it is hierarchical or co-operative caching, extending the notion of caching to streaming objects is not straightforward. The main reason for this is the size of streaming objects, which prevents them to be cached always in their entirety. Given that caches have finite disk space, it is not feasible to statically store more than a few complete streaming objects. If there are several simultaneous requests for different streaming objects, it is easy to show that the cache will be busy replacing one streaming object with another resulting in significant performance degradation.

Overall, it is not clear what is the best way of extending traditional caching for supporting streaming media. Straightforward extensions have major drawbacks and it is necessary to design more sophisticated mechanisms. In this work, we study three techniques to enhance caches to better support streaming media over the Internet. These include *Segmentation of Streaming Objects* into smaller units for caching purpose, *Placement and Replacement Policies* for streaming cache content, and *Cooperation of Caches* in the system. We call caching proxies specially designed for streaming media object *helpers* in the rest of the discussion.

4 Streaming Cache Placement and Replacement Policies

As with regular web objects, there are two integral parts in caching systems for streaming media: one is placement and replacement policies, the other is cache system organization and location of the proper caches to serve client requests. We look at the first issue in this section.

The design space for streaming cache placement and replacement policies can be explored in the following three dimensions: definition of hotness, data representation and data distribution.

4.1 Hotness Rating

Caching systems for streaming media objects aim for reducing end-to-end delay and reducing server load along with network load. This goal is achieved by distributing media objects among helpers located close to the clients. It is not feasible to replicate all media objects in all the helpers in the caching system due to limited disk space on helpers. To better utilize limited resources, it is natural to replicate only popular media objects.

From a helper's point of view, an object is hot if a large number of client requests arrive at this helper during short period of time. We define *helper hotness rating* of a media object served by the helper as follows:

$$\text{helper\_hotness\_rating} = \frac{\text{Total num of client requests}}{\text{Time span these requests are received at the helper}}$$

From the server's point of view, a media object is hot if there are a large number of requests for this object during short time period. However, the role of helpers is to prevent client requests from going to the server directly. The direct measurement of number of requests per unit time at the server does not make sense. Instead, each helper should report helper hotness rating for each media object to the server periodically, and the sum of helper hotness rating over all the helpers is the *server hotness rating.*

$$\text{server\_hotness\_rating} = \sum_{\text{all helpers}} \text{helper\_hotness\_rating}$$

When client request history is not long enough to calculate a reasonable server hotness rating, it makes sense to manually assign high ratings to some objects that are predicted to be very hot. For example, the top 10 videos of the week, CNN news during lunch hours, or the video on the NASA site from Mars landing.

Helper hotness rating is a local measure representing how popular an object is among clients served by this helper. However, the set of clients served by a particular helper could be changing dynamically depending on organization of helpers in the caching system.

It is not clear if client requests for media objects show spatial locality. In this context it means if an object is requested by a client. then it is likely to be requested by other clients close by.

If there is this kind of spatial locality, then it make sense to define a local-helper hotness rating as the average of helper hotness ratings over all neighboring helpers.

4.2 Data Representation: Entire Clip vs. Segmentation

We envision the number of streaming media objects on the web will grow with exponential speed in the near future. One major difference between streaming objects and regular objects on the web is their size. The size of streaming objects normally is at least an order or two larger than that of regular static web objects. For example, a single, two hour long MPEG movie requires about 1.4 Gbytes of disk space.

We have two design choices: one is to store video clips in their entirety on helpers, and treat them just as regular web objects; the other is to break video objects into smaller segments, and treat these segments as special web objects observing their temporal relationship.

Given a fixed investment on disk space, only a few video clips could be stored at a helper, thus, decreasing hit probability and efficiency of the caching system. It would be natural to adopt the second design to store videos in smaller segments. However for very hot videos, it still makes sense to store them in their entirety. This can be achieved by locking all the segments of popular videos on disk as a whole, or by making the segment size the same as the entire clip size. As storing the video in an all-or-nothing fashion is a special case of storing it in segments, we only use the second technique from now on.

Suppose the minimal allocation unit of static cache is a disk block of size S, then we can set the segment size to be any multiple of S. For the rest of the discussion, we assume a segment size of S. As a result, portions of a streaming object can be cached and replaced independently and therefore cache utilization is greatly increased.

There is a fundamental difference between segments of a streaming object and a set of static objects, such as HTML pages or images. Each stream segment cached has a starting playback time and an ending playback time. These segments are also inter-related by their starting and ending playback time. In addition, a client request for streaming objects contains explicit starting playback time and ending playback time. Thus, when a streaming request arrives at a cache, it is not simply the question of a hit or miss. Most likely the request will be a partial hit in the sense that one part of the requested streaming object is in the cache and the remaining parts are stored somewhere else. This is different from classical web caching. As a result, a requesting host might end up getting multiple pieces of the streaming object from different caches. This not only increases signaling cost, but also increases the probability of losing synchronization. Therefore, it is preferable to cache successive segments rather than caching a sequence of segments with multiple gaps in it.

To address this issue, we exploit the timing relationship between segments by using a fixed size *chunk* as a higher level caching unit. More precisely, we define the chunk size to be a multiple of S. Suppose we have an empty cache and begin caching the first chunk of a streaming object *chunk*[0]. As we receive the streaming data, disk blocks are allocated on demand to *chunk*[0]. When we reach the chunk boundary, we begin caching the next chunk, *chunk*[1]. In general, a chunk may not be cached to the chunk boundary because a stream can terminate any time. When subsequent streams of the same object are received, if possible, we continue filling existing partial chunks, and create new chunks as needed.

If we maintain that segments within each chunk are contiguous, then the number of gaps of a streaming object will be bounded. To achieve this, we follow the principle of *prefix caching* within each chunk. Under prefix caching, segments of a chunk are ejected from the end of the chunk by replacing the last segment first. We use a modified Least Recently Used (LRU) segment replacement policy with chunks as follows. (Other policies such as Least Frequently Used (LFU) can be applied. However, we do not explore the differences between replacement policies in this study.) We select the LRU chunk in the entire system. If this chunk is currently in use (a chunk is in use if any one of its segments is in use), then we select the next LRU chunk. If the selected chunk is not currently in use, we choose the *last* segment within this chunk to be the victim. LRU only determines which chunk is victimized, the actual replaced segment is chosen based on the prefix caching principle.

Note that the size and number of gaps in an object strongly depends on the chunk size. Large chunks increase the probability of blocking, while a small chunk size might increase the number of gaps. In any case, it might be necessary for clients to get multiple fragments from different helpers. This requires some form of intelligent pre-fetching of the following segments while a given segment is being streamed. Without such a technique, it is highly unlikely that the timing requirements can be met. This introduces a new dimension to caching when applied to streaming objects.

4.3 Data Distribution: Push, Pull or Demand Driven

As with regular web objects, placement policies for streaming objects also fall into the same three categories, the first is for the server to push popular streaming objects to helpers, the second is for each helper individually pulls popular objects from the server, and the third is demand driven. These techniques can be used independent of each other and we use use a combination of them.

4.3.1 Push from the Server

Studies have shown that access pattern for regular text and graphic web objects follows a Zipf like distribution [3]. Since we are not aware of any specific study on access pattern of streaming media on the web, it is reasonable to use the Zipf distribution as an approximation. Zipf's law states that for a given set of objects, the probability p that a request is for an object with rank r is inversely proportional to r. That is, $p = k(\frac{1}{r})$, where k is a constant.

The essential goal for a caching system is to distribute objects close to the requesting clients. If an object is popular, then requests for it are likely to come from many clients, and it is beneficial to distribute the objects to many helpers in the system. An intuitive approach is to distribute each object to a fraction of the helpers in the system according to the probability p that it is accessed. We call it *rank-based algorithm.*

Rank-Based Server-Push Algorithm: For a given set of streaming objects whose access patterns follow Zipf's law, each object has a rank r and the fraction of the helpers it is distributed to is $p = e(\frac{1}{r})$, where e is a constant. The fraction of helpers are randomly chosen because Zipf's law does not state which clients the requests for certain object are for.

However, this intuitive approach requires periodical recalculation of rank r of each object. Because rank r depends on two factors: one is the server hotness rating, the other is the total number of objects considered in the ranking. Any addition or deletion of the objects hosted by the server will result in changes in ranking and therefore results in changes in the fraction of the helpers they are distributed to.

To reduce the instability of the rank-based approach, we introduce the following *category-based algorithm.* A server keeps server hotness rating h_{server} for each streaming object it is hosting. The rating could either be manually assigned or collected from helpers in the system. Based on its hotness rating, the server assigns each object into one of the four categories: *cold*, *warm*, *hot*, and *very hot* as follows. Given three input hotness ratings $0 < R_1 < R_2 < R_3$,

$$\text{Object is} \begin{cases} \text{cold,} & \text{if } 0 \leq h_{server} < R_1 \\ \text{warm,} & \text{if } R_1 \leq h_{server} < R_2 \\ \text{hot,} & \text{if } R_2 \leq h_{server} < R_3 \\ \text{very hot,} & \text{if } R_3 \leq h_{server} \end{cases}$$

Category-Based Server-Push Algorithm: For objects in the same category, the server multicasts them to the same number of helpers as follows: Given four input parameters $0 \leq a_1 < a_2 < a_3 < a_4 \leq 1$,

$$\text{Object is multicast to} \begin{cases} a_1 \text{ of the helpers, if it is cold} \\ a_2 \text{ of the helpers, if it is warm} \\ a_3 \text{ of the helpers, if it is hot} \\ a_4 \text{ of the helpers, if it is very hot} \end{cases}$$

For example, setting $a_1 = 1/8$, $a_2 = 1/4$, $a_3 = 1/2$, and $a_4 = 1$ means multicasting very hot objects to all the helpers and hot objects to half of the helpers and so on. Again, the fraction of the helpers are randomly chosen from the set of all helpers.

Which category an object belongs to only depends on its server hotness rating. Therefore adding new objects at the server will not affect any existing object's category label and its distribution in the caching system.

Consider the disk space at all helpers as a whole, storing a streaming object in its entirety in $1/q$ of the helpers, where q is a positive integer, requires the same amount of disk space in the system as storing $1/q$ of the object in *all* the helpers. It is very common for users to start streaming video from its beginning then lose interest and tear down the connection. To better utilize disk resources, we can store only fractions of the streaming objects in the *fraction-based algorithm* as follows:

Fraction-Based Server-Push Algorithm: Server multicasts all the objects it is hosting to all the helpers. However, the fraction from the beginning of the object distributed is determined by the object's category label.

$$\begin{cases} a'_1 \text{ of each object is distributed, if it is cold} \\ a'_2 \text{ of each object is distributed, if it is warm} \\ a'_3 \text{ of each object is distributed, if it is hot} \\ a'_4 \text{ of each object is distributed, if it is very hot} \end{cases}$$

where $0 \leq a'_1 < a'_2 < a'_3 < a'_4 \leq 1$ are four input parameters.

For example, setting $a'_1 = 1/4$, $a'_2 = 1/3$, $a'_3 = 1/2$, and $a'_4 = 1$ implies very hot objects are distributed in their entirety, only the first half of each hot object is distributed, and so on.

The two approaches above can be applied at the same time as follows:

Category-and-Fraction-Based Server-Push Algorithm: Server multicasts fraction a' of each object to fraction a of the helpers where a and a' depend on the object's server hotness rating.

4.3.2 Pull by the Helpers

In the pull-by-helper approach, each helper independently requests objects from the server based on object popularity. Each cache keeps helper hotness rating h_{helper} for each streaming object requested by its clients. This rating could either be manually assigned or dynamically calculated as in Section 4.1. Each object falls into one of the four categories: cold, warm, hot, and very hot according to its h_{helper}. Each object potentially has two types of category labels: one is *server category label* stored at the server according to its server hotness rating, the other is *helper category label* kept at every helper according to its helper hotness rating at that particular helper. The latter is used in this approach.

Each helper requests only fraction of each object from the server in the fraction-based algorithm as follows:

Fraction-Based Helper-Pull Algorithm: The fraction from the beginning of the object requested is determined by the object's helper category label.

$$\text{A helper requests} \begin{cases} a'_1 \text{ of an object, if it is cold} \\ a'_2 \text{ of an object, if it is warm} \\ a'_3 \text{ of an object, if it is hot} \\ a'_4 \text{ of an object, if it is very hot} \end{cases}$$

where $0 \leq a'_1 < a'_2 < a'_3 < a'_4 \leq 1$.

For example, setting $a'_1 = 1/4$, $a'_2 = 1/3$, $a'_3 = 1/2$, and $a'_4 = 1$ implies very hot objects are requested in their entirety, only the first half of each hot object is requested, and so on.

4.3.3 Demand Driven

In the demand driven approach, clients request streaming objects from certain caches. The helpers that serve the requests cache the objects along with passing them to the clients. Since each object has a category label at the helper, the fraction-based algorithm can be used again.

Fraction-Based Demand-Driven Algorithm: The fraction from the beginning of the object cached at a helper is determined by the object's helper category label.

$$\text{A helper caches} \begin{cases} a'_1 \text{ of an object, if it is cold} \\ a'_2 \text{ of an object, if it is warm} \\ a'_3 \text{ of an object, if it is hot} \\ a'_4 \text{ of an object, if it is very hot} \end{cases}$$

where $0 \leq a'_1 < a'_2 < a'_3 < a'_4 \leq 1$.

4.4 Data Distribution: Deterministic vs. Random

Given that streaming objects are represented in chunks, and we use chunk as cache placement and replacement unit.(Footnote: On the top level we use chunk as cache placement and replacement unit. Within each chunk, we follow the prefix caching principle, that is, when a chunk is allocated, it is filled from the beginning as much as possible, and when a chunk is chosen to be replaced, its content is ejected from the end of the chunk on a segment-by-segment basis. Therefore, it is possible to have partial chunks. This reduces waste in disk space.) We propose two mechanisms to store chunks on disk: deterministically and randomly.

4.4.1 Deterministic Approach

In the deterministic approach, when a streaming media object is streamed to a helper, it is stored on disk chunk by chunk continuously from the beginning of the object. Since a stream can be torn down any time, the last chunk might not be full. The last access time for each streaming object is recorded. And if any chunk of an object is in use, the entire object is marked in use.

When there is not enough disk space available on the helper, one approach is to choose the Least Recently Used (LRU) object to be the victim, and overwrite its content chunk by chunk from the end of the object. Within each chunk, its content is replaced segment by segment from the end. If the LRU object is in use, then the second LRU object is chosen to be ejected chunk by chunk. The effect of this approach is that when a new stream comes in, it might replace older streaming objects one by one from the end.

Another approach is to choose a number of objects to form a *victim set*, and overwrite their content chunk by chunk from the end in a round-robin fashion. The goal is for a new stream to replace a set of older stream objects from the end simultaneously and leave the prefix of as many objects in cache as possible.

There are a number of ways to compose the victim set. One is to choose a number of objects that are least recently used. Another is to choose a set of objects whose helper hotness ratings are below certain threshold R. Yet another is to choose objects that are labeled either cold or warm by the helper.

When there is not enough disk space available and no chunks belonging to other objects can be ejected, the helper simply ignores the incoming stream.

Notice whenever a new streaming object is placed in the cache, it is allocated as much disk space as possible for the beginning fraction determined by the fraction-based algorithms. The cache replacement algorithm will replace it from the end. As a result, most of the time only the prefix of the objects are stored in the cache. An identical way of implementing this approach is to set the chunk size to be object size, and replace objects from the end segment by segment.

The advantage of this approach is each object is always stored in continuous fashion, there is no gap in any objects. The disadvantage is that if any chunk in an object is in use, then the entire object is in use and none of the blocks can be ejected from disk. It is very likely to result in the situation where all the streaming objects are in use and there is no space for the incoming stream. To alleviate this situation, instead of locking the entire object when any part of it is in use, we lock only the prefix of the object up-to the chunk that is in use. Therefore the tail of the object can be ejected to make space for new streams.

4.4.2 Random Approach

In the random approach, when a streaming media object is streamed to a helper, only certain chunks are selected randomly to be stored on disk. The last access time for each streaming object is recorded. And if any chunk of an object is in use, contrary to the deterministic approach, only this chunk is marked in use. Other chunks of the same object that are not in use therefore can be ejected.

The goal is to distribute chunks of the streaming media objects evenly in the system of helpers to avoid sending requests to the server as much as possible.

There are two kinds of media objects. For designated popular clips, they are expected to be popular in the future among all the clients. Therefore, we want to distribute their chunks evenly to all the helpers in the caching system. For media objects whose popularity is not known ahead of time and are only popular among a certain set of clients, we want to distribute their chunks to only a subset of the helpers that are close to the interested clients.

Random Cache Placement Policy Used in the Server Push Approach Assume for each designated popular clip, the sender multicasts the entire clip in the form of chunks to the group of all the helpers. However each helper does not cache the entire clip on its disk, it employs a random algorithm X to cache only certain chunks of the clip. Each helper keeps an estimate of the total number of helpers in the system k, and upon receipt of media object chunks, it only caches a fraction of them that is proportional to $1/k$. The algorithm is presented in Figure 1.

The total number of chunks to cache for each clip at each helper depends on the total number of helpers in the system. This helper number can either be set statically or measured dynamically as shown in Section 5.1. Assume at one helper, this estimation is k and N is the total number of chunks in the clip, the number of chunks this helper is caching is then $(a \times N)/k$.

Notice if the category-based server-push algorithm is used, then objects are multicast to a random

```
Input:  $k$ is the estimated size of helper caching system.
        $a$ is a constant and $1 \leq a < k$.

  $p = a/k$;
  Use IP address of the helper as the seed for a random number generator with uniform
  distribution between 0 and 1;

  Whenever a chunk arrives, generate a random number $r$;
      if ($0 < r < p$) then
          cache the chunk;
      else
          ignore the chunk;
      fi
```

Figure 1: Algorithm X for caching certain chunks of the clip

subset of the helpers, and k in Algorithm X should be a fraction of the helper system size.

Random Cache Placement Policy Used in Helper Pull and Demand Driven Approaches For other clips whose popularity is determined by clients' request pattern, whenever a media stream flows to a helper, this helper uses a random algorithm Y to cache only certain chunks of the clip. We define a *helper cluster* for a stream as the set of helpers that has requested and has cached the same media object. Each helper keeps an estimate of the size of the helper cluster k', and upon receipt of media object chunks, it only caches a fraction of them that is proportional to $1/k'$. Thus, with high probability the entire object is cached chunk by chunk in a subset of the helper caching system. Again, the total number of chunks to cache for a certain clip at each helper depends on the total number of helpers in the cluster. The algorithm is given in Figure 2.

The time t when the beginning of the stream arrives at each helper depends on user request pattern and therefore not correlated. Each helper uses t as the seed for its random number generator to make sure that each helper does not cache an identical sequence of chunks, and the chunks are evenly distributed among all the helpers requesting the media object.

The benefit of this scheme compared to the server-multicast-to-all-helpers scheme is that popular clips among certain clients are only distributed among helpers severing(and most likely are close to) these

```
Input:  $k'$ is the estimated size of the helper cluster.
        $t$ is the time when the beginning of the stream arrives at the helper.
        $a$ is a constant and $1 \leq a < k'$.

    $p = a/k'$;
    Use $t$ as the seed for a random number generator with normal distribution between 0 and 1;

    Whenever a chunk arrives, generate a random number $r$;
        if $(0 < r < p)$ then
            cache the chunk;
        else
            ignore the chunk;
        fi
```

Figure 2: Algorithm Y for caching certain chunks of the clip

clients.

It is not clear if client requests for media objects show spatial locality. In this context it means if an object is requested by a client, then it is likely to be requested by other clients close by. If this kind of spatial locality holds, then helpers inside a helper cluster are close to each other and it makes sense to distribute the entire clip chunk by chunk among different helpers inside a cluster.

Random Cache Replacement Policy If each helper has infinite disk space, following the two random cache placement mechanisms presented above, the chunks of clips will be evenly distributed among all the helpers or in some helper cluster. However, since helpers have limited disk space, a cache replacement policy is needed to guarantee that the chunks of clips are still evenly distributed when existing chunks in the cache are frequently replaced.

We propose a new cache replacement policy Z that combines Lease Recently Used (LRU) policy (or Least Frequently Used, LFU) with a randomized algorithm.

Each clip in a helper has an age factor indicating the most recent access time as in LRU. (for LFU, each helper keeps a frequency count.) Recall that each clip is divided into a number of chunks, and each helper has a subset of these chunks. If a chunk is requested by some client, then only the chunk is marked *in use*, and can not be ejected from disk.

When a new stream arrives at a helper, and new disk space is required, our random cache replacement policy Z is executed to decide which chunks to replace. First, we find a set of clips whose age (frequency if LFU is used) is above certain threshold. For each of these clips in the set, we replace their chunks randomly as chunks for the new stream are flushed to disk.

Notice the chunks in each old clip are randomly replaced instead of being replaced from the end. Therefore we still have an even distribution of the chunks of the clip among helpers. The more chunks from a clip are replaced, the more gaps and bigger gaps are potentially created in the clip. In the extreme case, all the chunks in the old clips are replaced by the chunks in the new clip that is streaming through the helper.

Modification on the distribution used to cache chunks Currently we use a uniform distribution to spread chunks of video clips over a set of caches. This means we envision the beginning of the clip and the end of the clip are accessed with equal probability. Normally this is not the case, when browsing, people are very likely to just watch the beginning of the clip and abort. So the prefix of video clips should be given more chance to be cached in helpers than the suffix. Therefore, we can use some distribution other than uniform distribution, for example, an exponential distribution, to assign prefix of the clip higher probability to be cached than its suffix.

4.4.3 Combination of Deterministic and Random Approach

A combination of deterministic and random approaches can be used in cache placement and replacement policies. A reasonable approach is to cache the entire clip chunk by chunk continuously when the clip is received by the helpers, and replace random not-in-use chunks from the LRU clip. This would also create gaps in the clip stored in cache.

5 Streaming Cache Cooperation

Existing web caching systems employs a statically configured hierarchy as in [2, 4, 8]. The obvious drawback of the scheme is the additional latency incurred in either going up the hierarchy or going to a sibling cache when the requested clip is stored in some other location in the hierarchy. This extra latency is eliminated when the request is sent directly to the helper with the clip cached. In order to achieve this, we propose a scalable state distribution algorithm that allows helpers to get an up-to-date view on what is cached on which helper in the system.

5.1 Mechanisms for Scalable State Distribution

State distribution aims to provide up-to-date information on the status of active helpers (Here active helpers mean helpers that are willing to help other helpers and clients). It allows helpers to identify other helpers and identify cached streaming objects at remote sites.

In our caching system design, streaming objects are stored in chunks. Under the deterministic cache placement and replacement policies, prefixes of clips are stored in continuous chunks as presented in Section 4.4.1. Therefore advertisement messages for state distribution purposes sent out from each helper should include for each streaming object the beginning and end time of the prefix. Under the random policies as given in Section 4.4.2, chunks stored on disk are not contiguous. Therefore for each object, there could be multiple beginning and end time pairs included in each advertisement message. The disadvantage of random policies is that the size of messages in state distribution grows linearly with the number of gaps in objects stored on disk.

A mechanism for scalable state distribution strives for two distinctive goals: network load should be kept as low as possible, and quality of state distribution should be as high as possible. Quality in state distribution is good if it allows helpers to get an up-to-date and a complete view on active helpers. However, these goals result in conflicting requirements. To achieve good quality in state distribution, it is necessary to send state updates as soon as there are any changes in helper states. This implies frequent sending of advertisement messages. In addition, quality of state distribution also depends on the visibility of helpers. Cooperation in a global environment is only possible when helpers are able to identify other helpers in a far distance. However, the visibility of helpers mainly depends on the scope of their advertisement messages. The larger the scopes of outgoing advertisement messages, the more clients will be able to identify a helper. From this point of view, using global scope for advertisements seems to be preferable. From a network's point of view, global flooding of advertisements is not a good idea. Rather, advertisements should be restricted to a local scope in order to minimize the aggregate network load. In addition, advertisements should be sent with low frequency to minimize control overhead.

Helpers are mainly interested in getting support from nearby helpers whose network distance is relatively small. Therefore, it is desirable to receive advertisements from nearby helpers (with respect to network distance) more frequently than from far away helpers. This allows finding an appropriate helper quickly while reducing the overhead caused by global advertisements.

Theses observations motivated the design of the *Expanding Ring Advertisement (ERA)*, which has originally been developed in the context of reliable multicast protocols [5]. The ERA algorithm makes use of TTL-based scope restriction. However, it could easily be modified to use administratively scoped multicast addresses or any other scoping mechanism. According to the ERA algorithm, Helpers send their advertisements with dynamically changing TTL values given in Table 1.

Interval No.	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
TTL	15	31	15	63	15	31	15	127	15	31	15	63	15	31	15	254	15	31

Table 1: TTL values used to send advertisements

The given TTL values are defined according to the scope regions in the current MBone. They can easily be adjusted to conform to specific network infrastructures. In the given scheme, the first message is sent with a scope of 15 (scope *local*), the second one with a value of 31 (scope *regional*), etc. The effect of this scheme is illustrated in Figure 3.

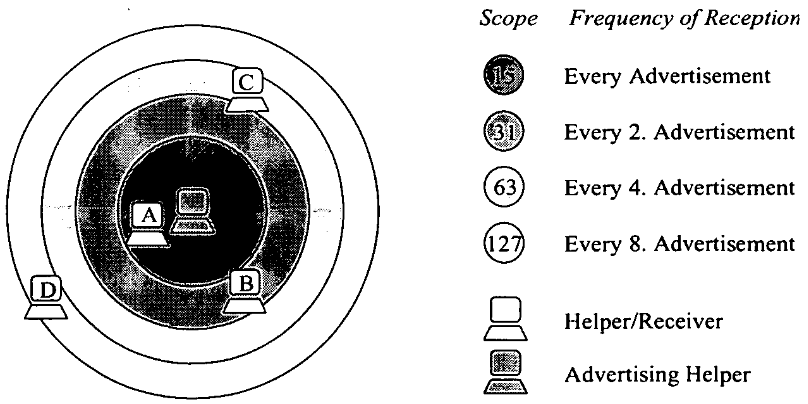

Figure 3: Different scoping areas

It shows that listening helpers within a scope of 15 (e.g. helper A in Figure 3) will get each of the advertisement messages (supposing there are no packet losses). If the distance between the advertising host and an active helper is between 16 and 31 (e.g. helper B in Figure 3), the helper will receive every second advertisement. This scheme continues in a way that every 16^{th} advertisement message will be distributed worldwide. The Expanding Ring Advertisement ensures that the frequency of advertisement messages exponentially decreases with increasing scope. Therefore, the scheme reduces network load while allowing short reaction time upon changes within a local scope. However, it still ensures global visibility of helpers in a larger time scale. Helpers are still able to identify other helpers in a far distance and to select them as secondary choices. Because advertisement messages are distributed to all the helpers periodically, each helper can estimate how many helpers are active in the system and how many helpers have a particular streaming object cached. This information is used in the random cache placement and replacement policies presented in Section 4.4.2.

The Expanding Ring Advertisement can also be used to estimate the number of hops between helpers without accessing raw IP packets.

5.2 Helper Selection

Upon receiving a request for a streaming object, a helper has to decide from where to get the requested data. It could get portions of or the entire object from its own local cache, directly from the server, or from the cache of any other helper that is able to serve the request. The decision making process is referred to as *helper selection.*

In general, a helper might get the requested data from multiple sources. It is also possible that no helpers are accessed and that the entire object is streamed from the server. It is the goal of the helper selection algorithm to find an optimal sequence of cache accesses in order to serve a request.

Basically, there are two ways to find such a sequence. First, the algorithm could determine all possible sequences and assess them before any data is streamed. Second, helper selection could use a step-wise algorithm to find one cache at a time, and finish streaming data from the selected cache before determining the next cache in sequence. The first solution requires some kind of look-ahead in time, which creates some problems in the context of streaming media. While the complete cache sequence has to be calculated in advance, content of caches might change over time. It might be the case that an optimal solution at calculation time will no longer be the best choice later on. In addition, users might interrupt data streaming at any time by performing VCR operations such as stop, pause, forward, and rewind. User behavior is not predictable, therefore re-calculation of the optimal sequence might become necessary. Considering all these aspects, we decide to select caches step-wise on demand. The following section explains the details of the selection algorithm.

Algorithm for Helper Selection During each iteration in the step-wise algorithm, a helper looks for the best cache present in the system to use in the step-wise algorithm . Once it chooses a cache, it gets as much data as there is in the selected cache up to the point where its own cache can be used again. The algorithm is outlined in Figure 4.

```
Input:   A playback request

tStart = start playback time of request;
done = false;
while not(done) do
   $S$ = the server;
   $C$ = list of all caches in known helpers with more than
   $M$ chunks of the streaming object after tStart;
   $soln$ = min_normalized_cost ($\bigcup(S, C)$);

          Request as much data from $soln$ as possible,
          up to the point where more data for the object exists in local cache;
          if (End of clip is reached) then
               done = true;
          else
               tStart = end playback time of request sent to $soln$;
               Sleep for the playback duration of request sent;
               continue;
          fi
od
```

Figure 4: Helper Selection Algorithm

Initially the playback starting time is the value requested by the client. During further iterations of the algorithm, the playback starting time advances to the end of the previous solution. Therefore, in general, the modified request in each iteration of the algorithm is from a certain starting time t_{start} to the end of the clip.

The algorithm finds the lowest cost cache currently among all the active helpers to serve the request using the function *min_normalized_cost.* Cost calculation in this function will be explained in the following section. For a cache to be chosen, it has to have more than M chunks of the streaming object after the requested playback time t_{start} where $M > 0$. This is to restrict the number of switches between caches. The disadvantages of frequent switching have been discussed in Section 4. If the cache found in this iteration is not the local cache, the helper gets as much data as possible from the remote cache, up to the point where its own local cache can be used, and advances the starting time t_{start} for the next iteration to the end of the current solution. The algorithm finds a minimum cost cache one step at a time. Therefore, it achieves a step-wise localized optimum.

Cost Function for Helper Selection Making use of a cache is associated with various cost factors. Accessing a cache might cause additional network load and increase the processing load at the caching helper. There is a large number of cost factors that might be considered. However, it is impossible to consider all of them without adding too much complexity to the system. Instead, a reasonable solution is to define a subset of useful indices and to specify a good heuristic to find an optimal solution according to user requirements.

Depending on the respective point of view, there are multiple goals for helper selection. While, for example, a network provider is mainly interested in optimizing resource utilization, a client wants to get immediate access to the requested service. These often competing goals lead to different cost functions, each of them focusing on different aspects of cost calculation. There are also cost functions that try to find a balance between various interests and cost factors. However, there is no “one size fits all” cost function that equally considers all interests and all cost factors.

We define one possible cost function that tries to find a balance between network and server/helper load associated with using a static or dynamic cache. Before we describe the function, we first specify the two cost factors that will be considered - network load and system load.

Network Load The network distance N between two helpers H_i and H_j is related to the number of hops on the path from H_i to H_j. Network distance is used as an approximation for the network load associated with data transmission from one host to another. The larger the network distance, the higher the aggregate network load for data transfers between these hosts.

In our scheme, helpers implicitly use the ERA algorithm for state distribution to estimate network distance between themselves. They classify remote helpers based on the observed scoping level and assign distance values to each of the classes according to Table 2.

class	ttl (scope)	network distance ($d(H_i, H_j)$)
local	0 - 15	1
regional	16 - 31	2
national	32 - 63	3
global	64 - 255	4

Table 2: Classes of network distance

For example, helper H_i classifies another helper H_j as *regional*, if it receives advertisement packets from H_j with a scope of 16 or greater, but no packets with a scope of 15 or smaller. In this case, the assigned network distance is 2. Accordingly, H_i classifies helpers within a scope of 32 to 63 *national*, and assigns network distance 3, and so forth. This mechanism is not restricted to the MBone scoping scheme. Instead, one might define different scoping levels according to the given network infrastructure.

System Load It is desirable to evenly distribute load among the helper machines and to avoid overloading popular helpers. The total number of incoming and outgoing streams is a good indicator of the current processing load at a system and is, therefore, included in the system's advertisement messages. Each system has a maximum number of streams it can serve concurrently. When the number of streams served is far less than this process capacity limit, serving one more stream has little impact on performance. However, when the system is operating close to its capacity, serving one more stream will degrade the performance significantly. As a result, the load cost should increase slightly for low load and increase sharply for high load. Therefore, the load cost L for requesting a stream from helper H (or server S) is defined as

$$L = \begin{cases} 1 & , \text{ if the cache is local} \\ \frac{maxLoad}{maxLoad - currentLoad} & , \text{ if the cache is remote} \end{cases}$$

where *currentLoad* is the current load at helper H and *maxLoad* the maximum allowable load for H. Notice that load cost is minimized for local cache because it does not incur any additional load at another helper or the server.

Calculating the Normalized Cost We define the normalized cost function for using a cache as the cost of getting a single segment from that cache. Using the two cost factors introduced above, the normalized cost is defined as $f = N \times L$, where N is the network cost (i.e. the network distance to the system holding the data) and L is the load cost (i.e. the processing load on the system holding the data).

It is important to note that, this cost function is minimized for the local cache solution, in which both N and L are one. As a result, whenever a local cache solution is available, we always make use of it to the fullest.

6 Conclusions and Future Work

Streaming media is becoming an increasingly more popular traffic type on the Internet. While streaming media objects usually incur much higher server and network loads than traditional Web objects and therefore can introduce more severe scalability bottleneck, surprisingly, little research has been done to study the issue of improving the scalability of streaming media over the Internet. This research is a first step to understand the issues of extending caches to support streaming media over the Internet.

References

[1] T Berners-Lee A. Lutonen, H.F. Nielsen. Cern httpd. http://www.w3.org/Daemon/Status.html, 1996.

[2] C.M. Bowman, P.B. Danzig, D.R. Hardy, U. Manber, M.F. Schwartz, and D.P. Wessels. Harvest: A scalable, customizable discovery and access system. Technical Report CU-CS-732-94, Dept. of Computer Science, University of Colorado, Boulder, USA, 1994.

[3] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker. Web caching and zipf-like distributions: Evidence and implications. In *Proceedings of IEEE Infocom'99*, March 1999. New York, NY, USA.

[4] A. Chankhunthod, P.B. Danzig, C. Neerdaels, M.F. Scwartz, and K.J. Worrell. A hierarchical internet object cache. In *Proceedings of the 1996 Usenix Technical Conference*, 1996.

[5] M. Hofmann and M. Rohrmuller. Impact of virtual group structure on multicast performance, 1998. Fourth International COST 237 Workshop, December 15-19, 1997, Lisboa, Portugal, Ed.: A. Danthine, C. Diot: From Multimedia Services to Network Services, Lecture Notes in Computer Science, No. 1356, Page 165-180, Springer Verlag, 1997.

[6] Real Networks. Internet homepage. http://www.real.com, 1999.

[7] Vxtreme. Internet homepage. http://www.vxtreme.com, 1999.

[8] D. Wessels. ICP and the squid cache. National Laboratory for Applied Network Research, 1999. http://ircache.nlanr.net/Squid.